**John Arena**

**Professor Gertner**

**CSC 342/343**

**Lab 1**

**Due 2/20/19**

**Spring 2019**

**Table of Contents**

**Note: Files sent to your email address on 12/3/18**

**Section 1) Objective pg. 3**

**Section 2) Description and Specification pg. 3**

**Section 3) Simulations pg. 12**

**Section 4) Conclusion pg. 26**

**Section 5) Appendix pg. 48**

**Section 6) Electric Layout pg. 55**

**Section 7) LTSPICE for Electric Layout pg. 71**

**Section 8) IRSIM for Electric Layout pg. 89**

**Section 9) Summary of Measurements pg. 96**

**Section 10) LTSPICE Comparisons of Schematic & Layout pg. 99**

**Section 11) Conclusion pg. 100**

**Section 12) References and Electric Files pg. 101**

**Section 1) Objective**

For this lab, the objective will be to apply everything we’ve learned about Quartus II in the tutorials. I will be doing the following:

* Building the following circuits using Object form and VHDL: 2to1 Multiplexer, 1-bit Half Adder, 1-bit Full Adder, 3to8 Decoder and 8to3 Encoder
* Verifying their correctness using waveform simulations
* Writing testbench files in VHDL to test the correct of the designs
* Programming pin assignments for the board

**Section 2) Description and Specifications**

2to1 Multiplexer

The first circuit I will be designing is a 2to1 **Multiplexer**. A multiplexer, also known as a mux, is “basically a switch that passes one of its data inputs through to the output, as a function of a set of select inputs”[1]. One of their uses is to choose among several multibit input numbers. The typical logic symbol of a 2to1 Multiplexer is shown below in Figure **NUM HERE**

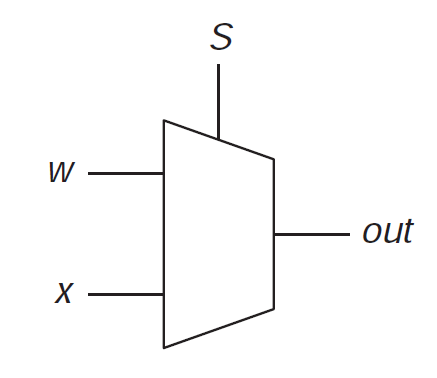


Figure NUM: 2to1 Multiplexer

The way a 2to1 multiplexer works as follows. If the select input, *S*, is equal to 0, the output, *out*, is equal to the value of X. If the select input is equal to 1, the output is equal to the value of Y. Table **NUM HERE** below shows the truth table of a 2to1 Mux. I will denote the two inputs as X and Y and the output as M.

|  |  |  |  |
| --- | --- | --- | --- |
| **X** | **Y** | **S** | **M** |
| **0** | **0** | **0** | **0** |
| **0** | **0** | **1** | **0** |
| **0** | **1** | **0** | **0** |
| **0** | **1** | **1** | **1** |
| **1** | **0** | **0** | **1** |
| **1** | **0** | **1** | **0** |
| **1** | **1** | **0** | **1** |
| **1** | **1** | **1** | **1** |

Table 1: 2to1 Multiplexer Truth Table

A simplified table is shown in Table **NUM HERE** below

|  |  |
| --- | --- |
| **S** | **M** |
| **0** | **X** |
| **1** | **Y** |

Table 2: 2to1 Multiplexer - Simplified Truth Table

We can derive the Boolean algebra expression of a 2to1 multiplexer from table **NUM BERE**. Looking at the table, we get the following

Equation 1: Out of 2to1 Mux

This can be composed of two AND gates, one OR gate and one NOT gate. The design of AND, OR and NOT gates can be designed using transistors, but are not within the scope of this course, so shall not be discussed.

The inputs and outputs will be assigned as follows on our board, seen in Figure **NUM HERE** below. It comes from the pin assignment text file for this circuit.

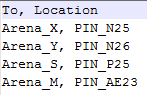


Figure 1: Pin Assignment for 2to1 Mux

The format is as follows. To, Location. To is the input/outputs from the object/vhdl file. The Location is the appropriate pins used for inputs and outputs. The pins are gotten from the **INSErT SOURCE HERE**.

On the next page in Figure **NUM HERE** is the design I made in Quartus for the 2to1 Multiplexer.

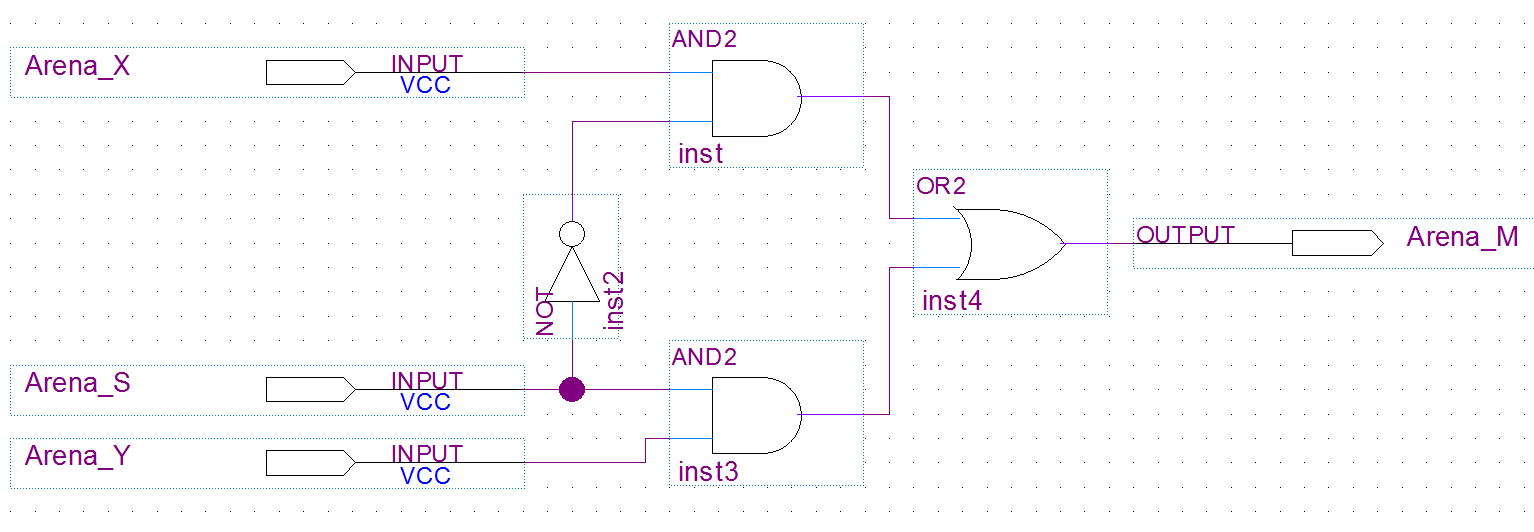


Figure 2: 2to1 Multiplexer at the Logic Level

As can be seen in the figure, the output consists of an 2-input OR gate connected to two 2-input AND gates with the appropriate inputs, with one AND gate having it’s input connected to a NOT gate just as described in Equation 1.

Below in Figure **NUM HERE** is the VHDL code for the circuit, generated by the MegaWizard tool.

-- (First, Last) John Arena - CSC 342/343 - Lab 1 - Spring 2019 Due: 2/20/19

-- megafunction wizard: %LPM\_MUX%

-- GENERATION: STANDARD

-- VERSION: WM1.0

-- MODULE: LPM\_MUX

-- ============================================================

-- File Name: Arena\_muxLPM.vhd

-- Megafunction Name(s):

-- LPM\_MUX

--

-- Simulation Library Files(s):

-- lpm

-- ============================================================

-- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

-- THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!

--

-- 13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition

-- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

--Copyright (C) 1991-2013 Altera Corporation

--Your use of Altera Corporation's design tools, logic functions

--and other software and tools, and its AMPP partner logic

--functions, and any output files from any of the foregoing

--(including device programming or simulation files), and any

--associated documentation or information are expressly subject

--to the terms and conditions of the Altera Program License

--Subscription Agreement, Altera MegaCore Function License

--Agreement, or other applicable license agreement, including,

--without limitation, that your use is for the sole purpose of

--programming logic devices manufactured by Altera and sold by

--Altera or its authorized distributors. Please refer to the

--applicable agreement for further details.

**LIBRARY** ieee**;**

**USE** ieee**.**std\_logic\_1164**.all;**

**LIBRARY** lpm**;**

**USE** lpm**.**lpm\_components**.all;**

**ENTITY** Arena\_muxLPM **IS**

**PORT**

**(**

Arena\_data0 **:** **IN** STD\_LOGIC **;** --Appropriate inputs/ outputs

Arena\_data1 **:** **IN** STD\_LOGIC **;** --for the external interface

Arena\_sel **:** **IN** STD\_LOGIC **;**

Arena\_result **:** **OUT** STD\_LOGIC

**);**

**END** Arena\_muxLPM**;**

**ARCHITECTURE** SYN **OF** arena\_muxlpm **IS**

-- type STD\_LOGIC\_2D is array (NATURAL RANGE <>, NATURAL RANGE <>) of STD\_LOGIC;

**SIGNAL** Arena\_sub\_wire0 **:** STD\_LOGIC\_VECTOR **(**0 **DOWNTO** 0**);** Various vars

**SIGNAL** Arena\_sub\_wire1 **:** STD\_LOGIC **;**

**SIGNAL** Arena\_sub\_wire2 **:** STD\_LOGIC **;**

**SIGNAL** Arena\_sub\_wire3 **:** STD\_LOGIC\_2D **(**1 **DOWNTO** 0**,** 0 **DOWNTO** 0**);**

**SIGNAL** Arena\_sub\_wire4 **:** STD\_LOGIC **;**

**SIGNAL** Arena\_sub\_wire5 **:** STD\_LOGIC **;**

**SIGNAL** Arena\_sub\_wire6 **:** STD\_LOGIC\_VECTOR **(**0 **DOWNTO** 0**);**

**BEGIN**

Arena\_sub\_wire4 **<=** Arena\_data0**;** --Appropriate assignments

Arena\_sub\_wire1 **<=** Arena\_sub\_wire0**(**0**);**

Arena\_result **<=** Arena\_sub\_wire1**;**

Arena\_sub\_wire2 **<=** Arena\_data1**;**

Arena\_sub\_wire3**(**1**,** 0**)** **<=** Arena\_sub\_wire2**;**

Arena\_sub\_wire3**(**0**,** 0**)** **<=** Arena\_sub\_wire4**;**

Arena\_sub\_wire5 **<=** Arena\_sel**;**

Arena\_sub\_wire6**(**0**)** **<=** Arena\_sub\_wire5**;**

LPM\_MUX\_component **:** LPM\_MUX

**GENERIC** **MAP** **(** -- Passing information to an entity

lpm\_size **=>** 2**,**

lpm\_type **=>** "LPM\_MUX"**,**

lpm\_width **=>** 1**,**

lpm\_widths **=>** 1

**)**

**PORT** **MAP** **(** --Port maps

data **=>** Arena\_sub\_wire3**,**

sel **=>** Arena\_sub\_wire6**,**

result **=>** Arena\_sub\_wire0

**);**

**END** SYN**;**

-- ============================================================

-- CNX file retrieval info

-- ============================================================

-- Retrieval info: PRIVATE: INTENDED\_DEVICE\_FAMILY STRING "Cyclone II"

-- Retrieval info: PRIVATE: SYNTH\_WRAPPER\_GEN\_POSTFIX STRING "0"

-- Retrieval info: PRIVATE: new\_diagram STRING "1"

-- Retrieval info: LIBRARY: lpm lpm.lpm\_components.all

-- Retrieval info: CONSTANT: LPM\_SIZE NUMERIC "2"

-- Retrieval info: CONSTANT: LPM\_TYPE STRING "LPM\_MUX"

-- Retrieval info: CONSTANT: LPM\_WIDTH NUMERIC "1"

-- Retrieval info: CONSTANT: LPM\_WIDTHS NUMERIC "1"

-- Retrieval info: USED\_PORT: data0 0 0 0 0 INPUT NODEFVAL "data0"

-- Retrieval info: USED\_PORT: data1 0 0 0 0 INPUT NODEFVAL "data1"

-- Retrieval info: USED\_PORT: result 0 0 0 0 OUTPUT NODEFVAL "result"

-- Retrieval info: USED\_PORT: sel 0 0 0 0 INPUT NODEFVAL "sel"

-- Retrieval info: CONNECT: @data 1 0 1 0 data0 0 0 0 0

-- Retrieval info: CONNECT: @data 1 1 1 0 data1 0 0 0 0

-- Retrieval info: CONNECT: @sel 0 0 1 0 sel 0 0 0 0

-- Retrieval info: CONNECT: result 0 0 0 0 @result 0 0 1 0

-- Retrieval info: GEN\_FILE: TYPE\_NORMAL Arena\_muxLPM.vhd TRUE

-- Retrieval info: GEN\_FILE: TYPE\_NORMAL Arena\_muxLPM.inc FALSE

-- Retrieval info: GEN\_FILE: TYPE\_NORMAL Arena\_muxLPM.cmp TRUE

-- Retrieval info: GEN\_FILE: TYPE\_NORMAL Arena\_muxLPM.bsf FALSE

-- Retrieval info: GEN\_FILE: TYPE\_NORMAL Arena\_muxLPM\_inst.vhd FALSE

-- Retrieval info: LIB\_FILE: lpm

Figure 3 : VHDL Code for 2to1 Mux

1-bit Half Adder

The second circuit I will be designing is a **1-bit Half Adder.** A 1-bit Half Adder is a circuit that can add two 1-bit numbers, and produce a Sum and a possible carry over. We know from basic math for example, if we had 3+3, we will have a sum of 6 with a carry over of 0, but 7+4 will produce a sum off 11 with a carry over of 1. We know from Boolean algebra that a 1-bit number is base two, since it can only have two digits, either a 0 or a 1. Since that is the case, let’s quickly review the rules of binary addition of 1-bit numbers

Knowing these rules, below in Table **NUM** describes the functionality of a 1-bit half adder.

|  |  |  |  |
| --- | --- | --- | --- |
| **X** | **Y** | **Sum** | **Carry Out** |
| **0** | **0** | **0** | **0** |
| **0** | **1** | **1** | **0** |
| **1** | **0** | **1** | **0** |
| **1** | **1** | **0** | **1** |

Table 3: 1-bit Half Adder Truth Table

We can derive the Boolean algebra expression of a 1-bit adder from table **NUM BERE**. Looking at the table, we get the following in equation **NUM HERE** below.

Equation 1: Out of 2to1 Mux

Looking at these functions, we can see Sum is an XOR function of variables X and Y, while Carry Out is an AND function of X and Y. These will be created using an XOR gate and an AND gate.

The inputs and outputs will be assigned as follows on our board, seen in Figure **NUM HERE** below. It comes from the pin assignment text file for this circuit.

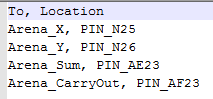


Figure 4: Pin Assignment for 2to1 Mux

There is 2 input switches used and 2 output LEDs. Below in Figure **NUM HERE** is the design I made in Quartus for the 1-bit Half-Adder.

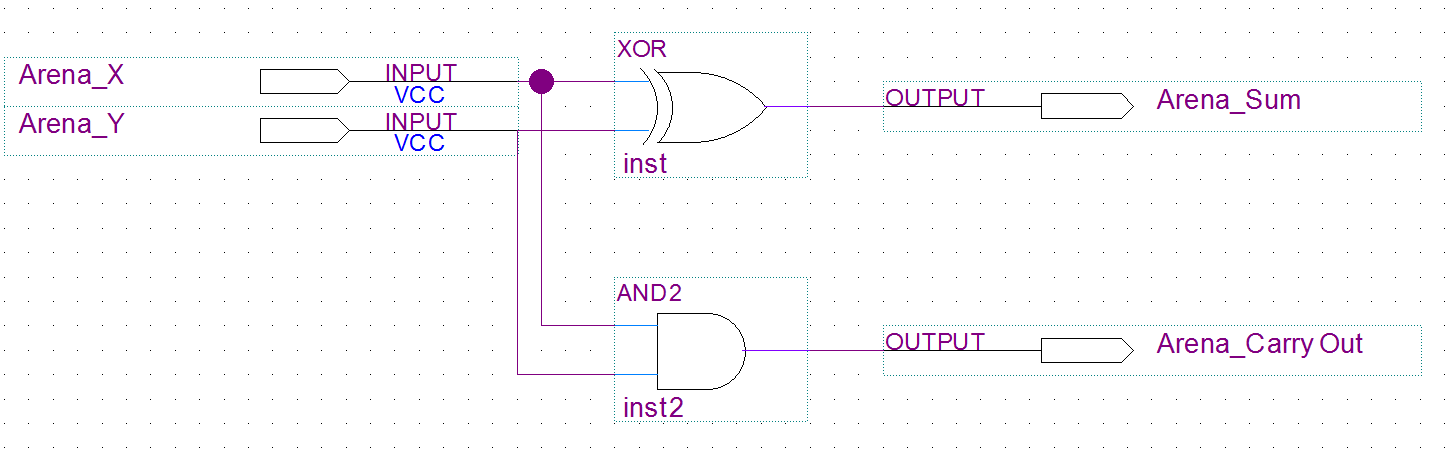


Figure 5: 1-bit Half Adder at Gate Level

As seen in the figure, there are two inputs, X and Y going into the XOR gate with the output as the Sum, and X and Y going into an AND gate with the output as CarryOut.

On the next page in Figure **NUM HERE** is the VHDL code I created for the 1-bit Half Adder.

-- (First, Last) John Arena - CSC 342/343 - Lab 1 - Spring 2019 Due: 2/20/19

-- Arena\_HalfAdder.vhd

**library** ieee**;**

**use** ieee**.**std\_logic\_1164**.all;**

**Entity** Arena\_HalfAdder **is**

**Port(**

Arena\_X**,** Arena\_Y**:** **in** std\_logic**;** -- Two Inputs for X and Y

Arena\_Sum**,** Arena\_CarryOut**:** **out** std\_logic -- Two outputs for the Sum and the CarryOut bit.

**);**

**end** Arena\_HalfAdder**;** -- End of Entity Arena\_HalfAdder

**Architecture** Arena\_Arch\_HalfAdder **of** Arena\_HalfAdder **is** -- Architecture of the Entity (Describes functionality)

**begin**

Arena\_Sum **<=** **(**Arena\_X xor Arena\_Y**);** -- Sum = X XOR Y

Arena\_CarryOut **<=** **(**Arena\_X and Arena\_Y**);** -- Carryout = X\*Y

**end** Arena\_Arch\_HalfAdder**;** -- End of Architecture statement

Figure 6: 1-bit Half Adder VHDL Code

1-bit Full Adder

The third circuit I will be designing is a **1-bit Full Adder.** A 1-bit Half Adder is a circuit that has three inputs. Two 1-bit numbers and one CarryIn input and produces a sum and a possible carry over. We know from basic math for example, if we have 17+14, we get 31, with a carry over and the tenth’s place gets a CarryIn of 1. We know from Boolean algebra that a 1-bit number is base two, since it can only have two digits, either a 0 or a 1. Since that is the case, let’s quickly review the rules of binary addition of 1-bit numbers

Knowing these rules, below in Table **NUM** describes the functionality of a 1-bit half adder.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **X** | **Y** | **Carry In** | **Sum** | **Carry Out** |
| **0** | **0** | **0** | **0** | **0** |
| **0** | **0** | **1** | **1** | **0** |
| **0** | **1** | **0** | **1** | **0** |
| **0** | **1** | **1** | **0** | **1** |
| **1** | **0** | **0** | **1** | **0** |
| **1** | **0** | **1** | **0** | **1** |
| **1** | **1** | **0** | **0** | **1** |
| **1** | **1** | **1** | **1** | **1** |

Table 3: 1-bit Half Adder Truth Table

We can derive the Boolean algebra expression of a 1-bit Full Adder from table **NUM BERE**. Looking at the table, we get the following in equation **NUM HERE** below.

Equation 1: Out of 2to1 Mux

Looking at these equations, it can be seen it can be derived to much simpler equations using Boolean algebra,. What’s interesting is as follows. Taking a look at the Sum equation, if we say for example **(X xor Y)** can be represented by **M,** we can say **Sum = Ci xor M.** This looks like what we had for the 1-bit Half Adder in equation **NUM HERE**. For the Carry Out, notice we can also say **Ci(M)**, which is also from the 1-bit Half Adder. So I can design this full adder by cascading two half adders.

The inputs and outputs will be assigned as follows on our board, seen in Figure **NUM HERE** below. It comes from the pin assignment text file for this circuit.

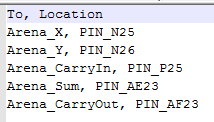


Figure 4: Pin Assignment for 2to1 Mux

There is 3 input switches used and 2 output LEDs. Before showing the 1-bit Full Adder, below in Figure **NUM HERE** is my symbol I created for a 1-bit Half Adder.

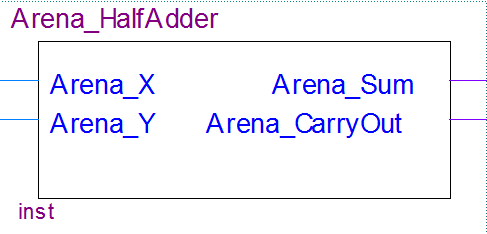


Figure 5: 1-bit Half Adder Symbol

Below in Figure **NUM HERE** is the 1-bit Full Adder using Half Adders.

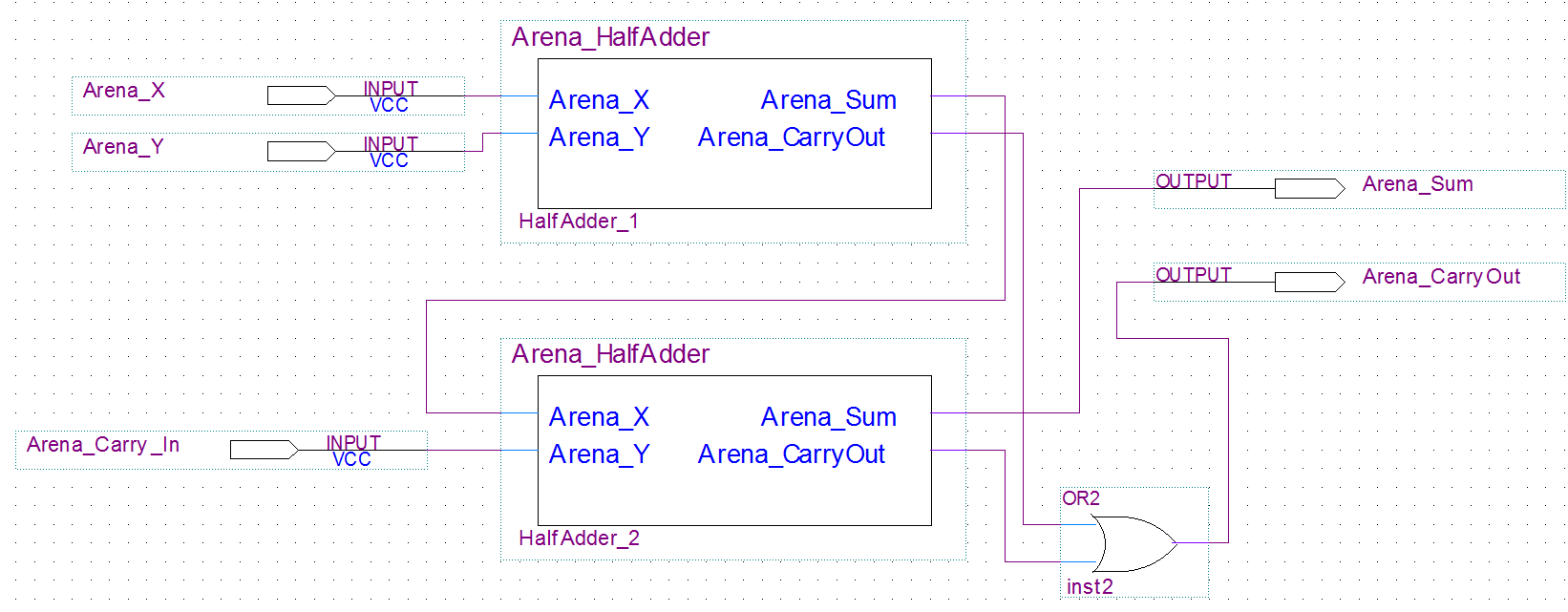


Figure 5: 1-bit Full Adder

As seen in the figure, there are two half adder’s cascaded together.

Looking at Arena\_X for Half Adder 2, it takes in the output of the Sum in Half Adder 1, which is **X xor Y**. Arena\_Y for Half Adder 1 takes in the CarryIn input. Knowing the design of the half adders from **Fig num here**, Arena\_Sum for Half Adder 2 will be **Sum = Ci xor (X xor Y).**

Since for the CarryOut of Half Adder 2 is **,** we know that the CarryOut of Half Adder 1 is (**XY**), so this can be plugged into an OR gate with the output of the Carryout of Half Adder 2, which gives us the final CarryOut of the Full Adder.

On the next page in figure **NUM HERE** is the VHDL code I created for the 1-bit Full Adder.

-- (First, Last) John Arena - CSC 342/343 - Lab 1 - Spring 2019 Due: 2/20/19

-- Arena\_FullAdder.vhd

**library** ieee**;**

**use** ieee**.**std\_logic\_1164**.all;**

**entity** Arena\_FullAdder **is**

**port(**

Arena\_X**,** Arena\_Y**,** Arena\_CarryIn **:** **in** std\_logic**;** -- Three inputs, X, Y and CarryIn

Arena\_Sum**,** Arena\_CarryOut **:** **out** std\_logic -- Two outputs, Sum and CarryOut

**);**

**end** Arena\_FullAdder**;** -- End of entity

**architecture** Arena\_Arch\_FullAdder **of** Arena\_FullAdder **is** -- Architecture describing functionality

**begin**

Arena\_Sum **<=** Arena\_CarryIn xor Arena\_X xor Arena\_Y**;** -- Sum = Ci xor (X xor Y)

Arena\_CarryOut **<=** **(**Arena\_X and Arena\_Y**)** or **(**Arena\_CarryIn and **(**Arena\_X xor Arena\_Y**));** -- Co (XY) or (Ci(X xorY))

**end** Arena\_Arch\_FullAdder**;** -- end of architecture

Figure 6: 1-bit Full Adder VHDL Code

3to8 Decoder

The fourth circuit I will be designing is a **3 to 8 Decoder.**  A decoder, also known as a binary decoder, “is a device that, when activated, selects one of several output lines, based on a coded input signal. Most commonly, the input is an n-bit binary number, and there are up to 2^n output lines.” “The inputs are treated as a binary number, and the output selected is made active”. There are two forms of decoders. One is called *active high*, and one is called *active low.* Active high means an active output is 1 and an inactive output is 0. Active low is the opposite, where an active output is 0 and an inactive output is 1. For my design, I will be using an active high.

As the name states, this is a 3to8 decoder, meaning a 3 bit number to 8 corresponding output lines. As the definition even said, with n=3, 2^3 = 8. With that said, I came up with the truth table in table **NUM HERE** below. The 3 inputs shall be denoted **A, B, C** and the outputs as **F0, F1…F7** for a total of 8.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **A** | **B** | **C** | **F0** | **F1** | **F2** | **F3** | **F4** | **F5** | **F6** | **F7** |
| **0** | **0** | **0** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| **0** | **0** | **1** | **0** | **1** | **0** | **0** | **0** | **0** | **0** | **0** |
| **0** | **1** | **0** | **0** | **0** | **1** | **0** | **0** | **0** | **0** | **0** |
| **0** | **1** | **1** | **0** | **0** | **0** | **1** | **0** | **0** | **0** | **0** |
| **1** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **0** |
| **1** | **0** | **1** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** |
| **1** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **0** |
| **1** | **1** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** |

Table 3: 3to8 Decoder Truth Table

We can derive the Boolean algebra expression of each active high. They are as follows

Looking at these equations, it can be seen these are all 3 input AND gates with NOT gates. So the design will consist of 8 three-input AND gates and 3 NOT gates.

The inputs and outputs will be assigned as follows on our board, seen in Figure **NUM HERE** below. It comes from the pin assignment text file for this circuit seen below in Figure **NUM here**.

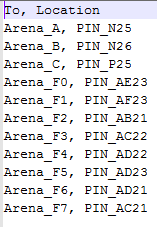


Figure 4: Pin Assignment for 2to1 Mux

There is 3 input switches used and 8 output LEDs. On the next page in figure **NUM HERE** is the design of the circuit.

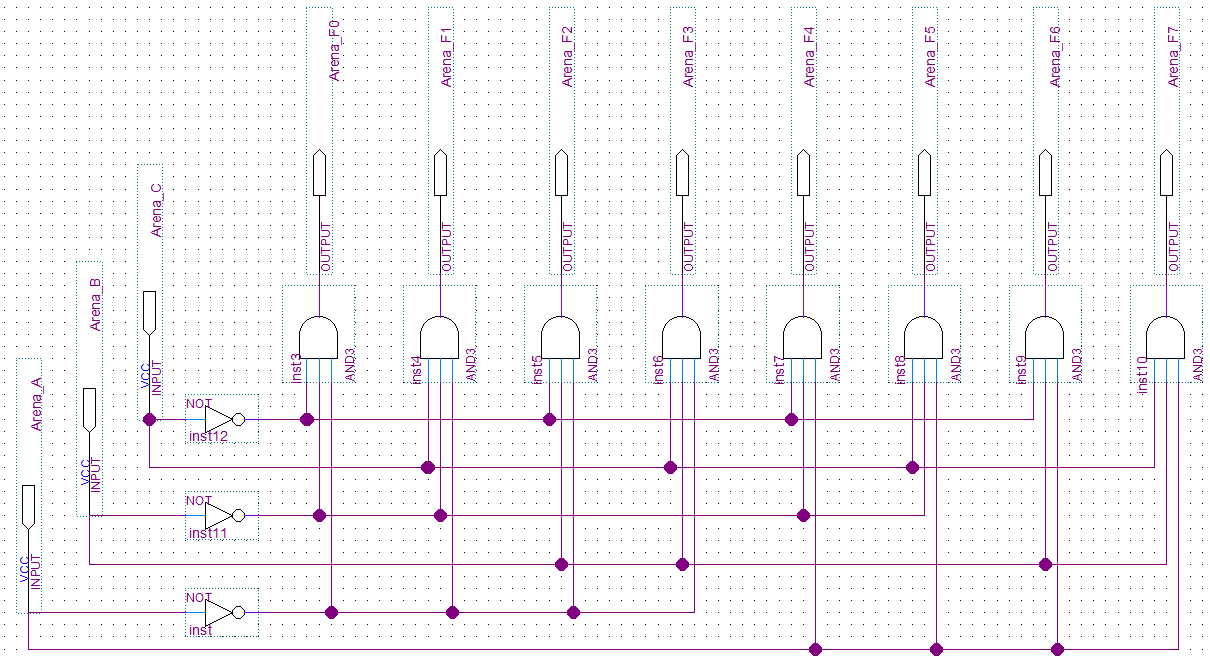


Figure 5: 3to8 Decoder Gate Level

Below in figure **NUM HERE** is the VHDL code I created for the 3to8 Decoder.

-- (First, Last) John Arena - CSC 342/343 - Lab 1 - Spring 2019 Due: 2/20/19

-- Arena\_3to8Decoder.vhd

**library** ieee**;**

**use** ieee**.**std\_logic\_1164**.all;**

**entity** Arena\_3to8Decoder **is** -- Decoder entity

**port(**

Arena\_A**,** Arena\_B**,** Arena\_C **:** **in** std\_logic**;** -- 3 inputs, A B and C

Arena\_F0**,** Arena\_F1**,** Arena\_F2 **:** **out** std\_logic**;** -- 8 outputs, F0..F7

Arena\_F3**,** Arena\_F4**,** Arena\_F5 **:** **out** std\_logic**;**

Arena\_F6**,** Arena\_F7 **:** **out** std\_logic -- all the way up to F7

**);**

**end** Arena\_3to8Decoder**;** -- end of entity

**architecture** Arena\_Arch\_3to8Decoder **of** Arena\_3to8Decoder **is** -- Architecture of Decoder, describing functionality

**begin**

Arena\_F0 **<=** '1' **when** **(**Arena\_A **=** '0' and Arena\_B **=** '0' and Arena\_C **=** '0'**)** **else** '0'**;** -- Set high when appropriate

Arena\_F1 **<=** '1' **when** **(**Arena\_A **=** '0' and Arena\_B **=** '0' and Arena\_C **=** '1'**)** **else** '0'**;** -- Otherwise 0

Arena\_F2 **<=** '1' **when** **(**Arena\_A **=** '0' and Arena\_B **=** '1' and Arena\_C **=** '0'**)** **else** '0'**;**

Arena\_F3 **<=** '1' **when** **(**Arena\_A **=** '0' and Arena\_B **=** '1' and Arena\_C **=** '1'**)** **else** '0'**;**

Arena\_F4 **<=** '1' **when** **(**Arena\_A **=** '1' and Arena\_B **=** '0' and Arena\_C **=** '0'**)** **else** '0'**;**

Arena\_F5 **<=** '1' **when** **(**Arena\_A **=** '1' and Arena\_B **=** '0' and Arena\_C **=** '1'**)** **else** '0'**;**

Arena\_F6 **<=** '1' **when** **(**Arena\_A **=** '1' and Arena\_B **=** '1' and Arena\_C **=** '0'**)** **else** '0'**;**

Arena\_F7 **<=** '1' **when** **(**Arena\_A **=** '1' and Arena\_B **=** '1' and Arena\_C **=** '1'**)** **else** '0'**;**

**end** Arena\_Arch\_3to8Decoder**;** -- end of architecture

Figure 6: 3to8 Decoder VHDL code

8to3 Encoder

The fifth and final circuit I will be designing is a **8 to 3 Encoder.**  A decoder, also known as a *binary encoder* is essentially the inverse of a binary decoder. So essentially whatever is active high (assuming this is an active high encoder), it will produce a 3-bit binary number output. “It is useful when one of several devices may be signaling a computer (by putting a 1 on a wire from that device); the encoder then produces the device number).” For my design, I will be using an active high.

As the name states, this is a 8to3 Encoder, meaning an 8-bit input to a 3-bit output.. We know a 3-bit number has 2^3=8 numbers, 0-7, so we need 8 states, so 8 possible inputs. With that said, I came up with the truth table in table **NUM HERE** below. The 8 inputs shall be denoted **Y0, Y1…Y7** and the output as **A, B and C.**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Y0** | **Y1** | **Y2** | **Y3** | **Y4** | **Y5** | **Y6** | **Y7** | **F0** | **F1** | **F2** |
| **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **0** |
| **0** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **0** | **1** |
| **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **0** | **1** | **0** |
| **0** | **0** | **0** | **0** | **1** | **0** | **0** | **0** | **0** | **1** | **1** |
| **0** | **0** | **0** | **1** | **0** | **0** | **0** | **0** | **1** | **0** | **0** |
| **0** | **0** | **1** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **1** |
| **0** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **1** | **0** |
| **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **1** | **1** |

Table 3: 8to3 Encoder Truth Table

We can derive the Boolean algebra expression of each output. They are as follows

Looking at these equations, it can be seen these are all 4 input OR gates. The design will consist of 3 OR gates.

The inputs and outputs will be assigned as follows on our board, seen in Figure **NUM HERE** below. It comes from the pin assignment text file for this circuit seen below in Figure **NUM here**.

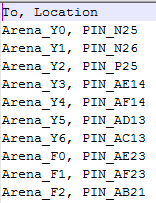


Figure 4: Pin Assignment for 8to3 Encoder

There is 8 input switches used and 3 output LEDs. On the next page in figure **NUM HERE** is the design of the circuit.

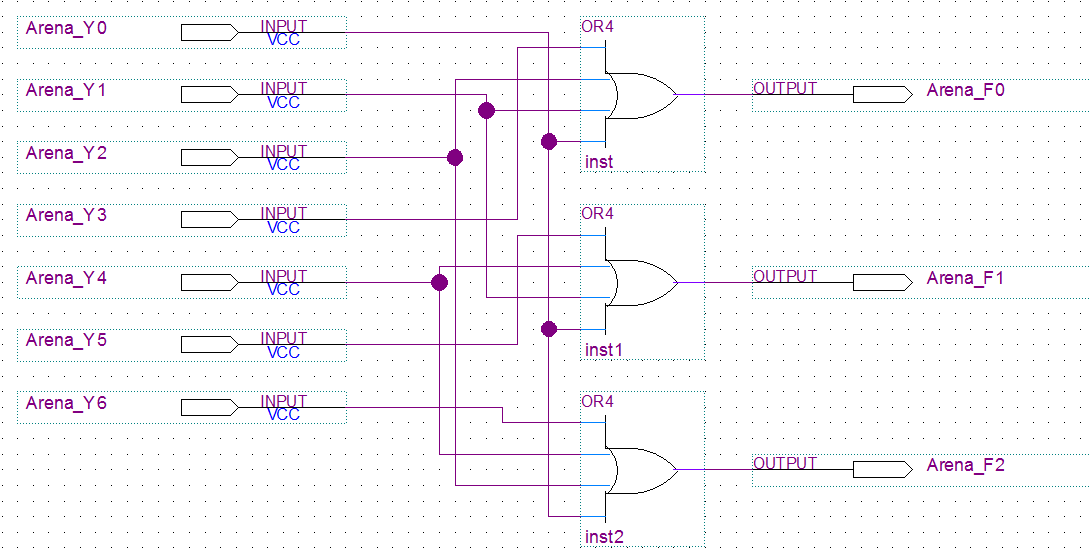


Figure 5: 8to3 Encoder at Gate Level

Below in figure **NUM HERE** is the VHDL code I created for the 8to3 Encoder.

-- (First, Last) John Arena - CSC 342/343 - Lab 1 - Spring 2019 Due: 2/20/19

-- Arena\_3to8Decoder.vhd

**library** ieee**;**

**use** ieee**.**std\_logic\_1164**.all;**

**entity** Arena\_8to3Encoder **is** -- Entity for Encoder

**port(**

Arena\_Y0**,** Arena\_Y1**,** Arena\_Y2**,** Arena\_Y3 **:** **in** std\_logic**;** -- 8 inputs

Arena\_Y4**,** Arena\_Y5**,** Arena\_Y6**,** Arena\_Y7 **:** **in** std\_logic**;**

Arena\_F0**,** Arena\_F1**,** Arena\_F2 **:** **out** std\_logic -- 3 outputs

**);**

**end** Arena\_8to3Encoder**;** -- End of entity

**architecture** Arena\_Arch\_8to3Encoder **of** Arena\_8to3Encoder **is** --Architecture to describe functionality

**begin**

Arena\_F0 **<=** **(**Arena\_Y3 or Arena\_Y2 or Arena\_Y1 or Arena\_Y0**);** -- F0 = Y3+Y2+Y1+Y0

Arena\_F1 **<=** **(**Arena\_Y5 or Arena\_Y4 or Arena\_Y1 or Arena\_Y0**);** -- F1 = Y5+Y4+Y1+Y0

Arena\_F2 **<=** **(**Arena\_Y6 or Arena\_Y4 or Arena\_Y2 or Arena\_Y0**);** -- F2 = Y6+Y4+Y2+Y0

**end** Arena\_Arch\_8to3Encoder**;** -- End of architecture

Figure 6: 8to3 Encoder VHDL code

**Section 3) Simulations**